Lecture Participation Poll #13

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 13: Multifile C Management

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

Assignments

HW2 Live - Soft Deadline Thursday October 29th at 9pm PST

- Autograder updated

HW3 coming later today – last assignment before midpoint deadline

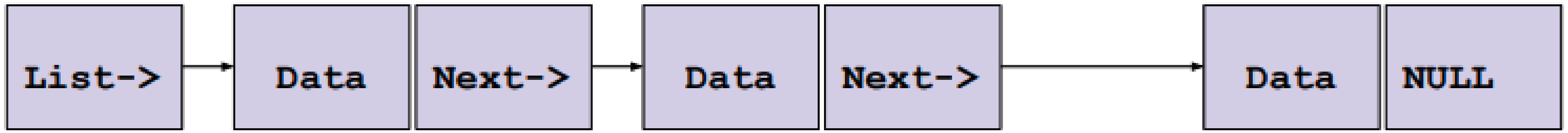
Reminder: Midpoint Deadline Friday November 6th at 9pm PST

Review Assignment Live – Due Wednesday

- 24 hrs late 20% penalty
- 48 hrs late 50% penalty
- Not accepted more than 48hrs late

Student survey: Week student survey

Linked Lists




```c
#include <stdlib.h>
#include <stdio.h>

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next) {
    Node *node = (Node*)malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

```c
int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    printf(
        "%d%d%d\n",
        n3->value,
        n3->next->value,
        n3->next->next->value
    );

    free(n3);
    free(n2);
    free(n1);
}
```

Multi-File C Programming

- You can split C into multiple files!
  - What if we wanted to use Linked List code in a different project?
  - If the linked list code is long, it can make files unwieldy
  - What if we want to separate our "main" from the struct definitions
- Pass all ".c" files into gcc:

```
gcc -o try_lists ll.c main.c
```

Must include code header files to enable one file to see the other, otherwise you have linking errors

```
$ gcc -g -Wall -o try_lists ll.c main.c
main.c: In function 'main':
main.c:5:5: error: unknown type name 'Node'
    5 |     Node *n1 = make_node(4, NULL);
      |     ^~~~
main.c:5:16: warning: implicit declaration of function 'make_node' [-Wimplicit-function-declaration]
    5 |     Node *n1 = make_node(4, NULL);
      |                ^~~~~~~~~
```

Sharing code across files

- Must always declare a function or struct in every file it's used in
  - Thank goodness C lets us separate declarations and definitions ;)
  - Include function header as definition

```
Node *make_node (int value, Node *next);
```

  - Include struct type definition

```
typedef struct Node
{
    int value;
    struct Node *next;
} Node;
```

```
#include <stdlib.h>
#include <stdio.h>

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next);

int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    // rest of main…
}
```

main.c

```
#include <stdlib.h>

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next);

Node *make_node(int value, Node *next) {
    Node *node = (Node*)malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

ll.c

Header Files

- Copying your function declarations to every file you want to use them is not fun
  - If you forget to make a change to all of them, confusing errors occur!
- A **header file** (.h) is a file which contains just *declarations*
- #include inserts the contents of a header file into your .c file
  - Put declarations in a header, then include it in all other files
  - Two types of #include

```
#include <stdio.h>
```

  - Used to include external libraries. Does not look for other files that you created.

```
#include "myfile.h"
```

  - Used to include your own headers. Searches in the same folder as the rest of your code.

ll.h

```c
typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next);
```

ll.c

```c
#include <stdlib.h>
#include <stdio.h>

#include "ll.h"

Node *make_node(int value, Node *next) {
    Node *node = (Node*)malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

main.c

```c
#include "ll.h"

int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    // rest of main…
}
```

Header Guards

- Consider the following header structure:
  - Header A includes header B.
  - Header C includes header B.
  - A source code file includes headers A and C.
  - The code now includes two copies of header B!
  - Solution: "**header guard**"
    - Uses ifndef to check if header is already defined for this file

```
#ifndef LL_H
#define LL_H

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next);
#endif
```

ll.h

```
#include <stdlib.h>
#include <stdio.h>

#include "ll.h"

Node *make_node(int value, Node *next) {
    Node *node =
(Node*)malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

ll.c

```
#include "ll.h"

int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    // rest of main…
}
```

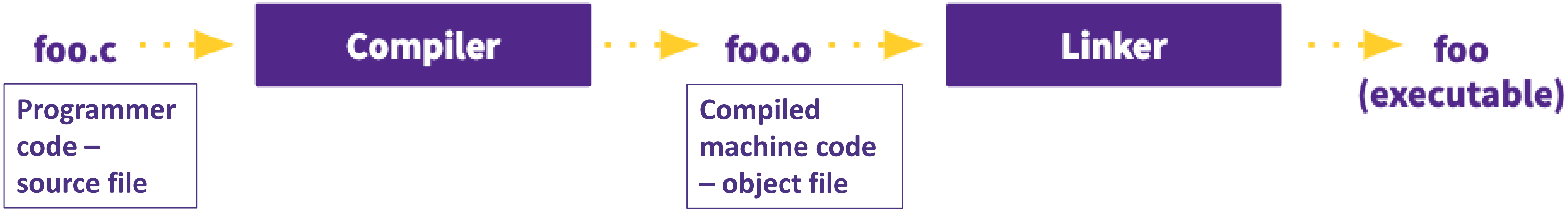
main.c

Libraries & Object Files in C

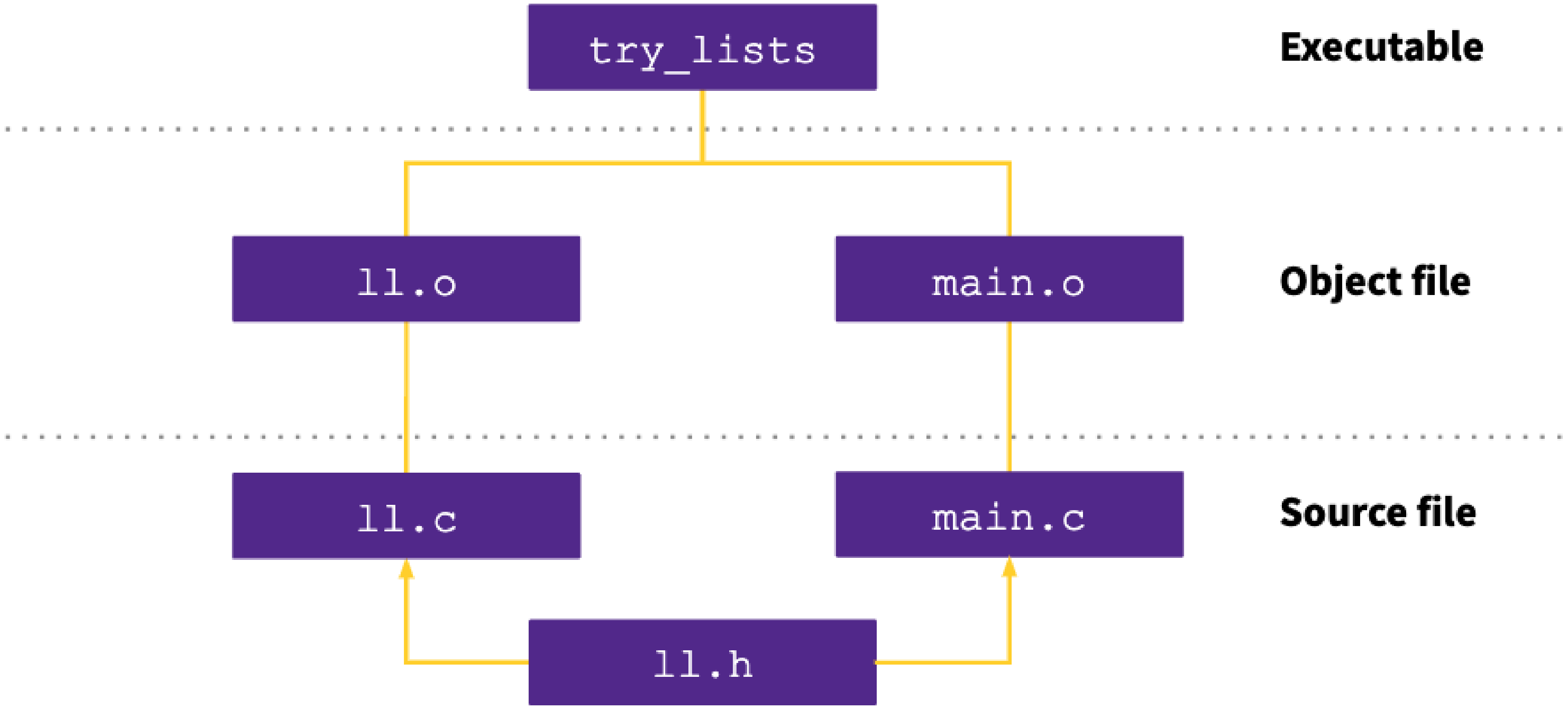
- Remember `#include <stdio.h>`?
- Tells our .c file what function declarations are in `stdio.h`
  - but what about the function definitions? (i.e. the code)
  - We don't have access to `stdio.c`
- Instead, we have a pre-compiled library that contains the function definitions
  - The stdio library is included by default with gcc
- In C, these "libraries" are called object files
  - Object files contain the machine code for the functions within
  - When compiled, a function is turned into "machine code" which the physical CPU electronics can understand

Linking in C

- Every time you have compiled something with gcc, you have actually been doing *two* things:
  - Compiling: Translating C code (a single .c file) into machine code stored in object files
  - Linking: Combining many object files into one executable
- Why separate these two?
  - Compile each object once and re-use it for multiple executables
  - Building multiple programs which use some of the same source code doesn't require recompilation
  - incremental compilation: Huge projects can take hours or days to compile from scratch! We can save time by only re-compiling what has changed.
    - Slow-to-compile files which you don't change often don't have to be re-compiled



Dependency Tree: linked list project



Example

Consider this dependency graph. What files (source and object) are required when building `program_two`?

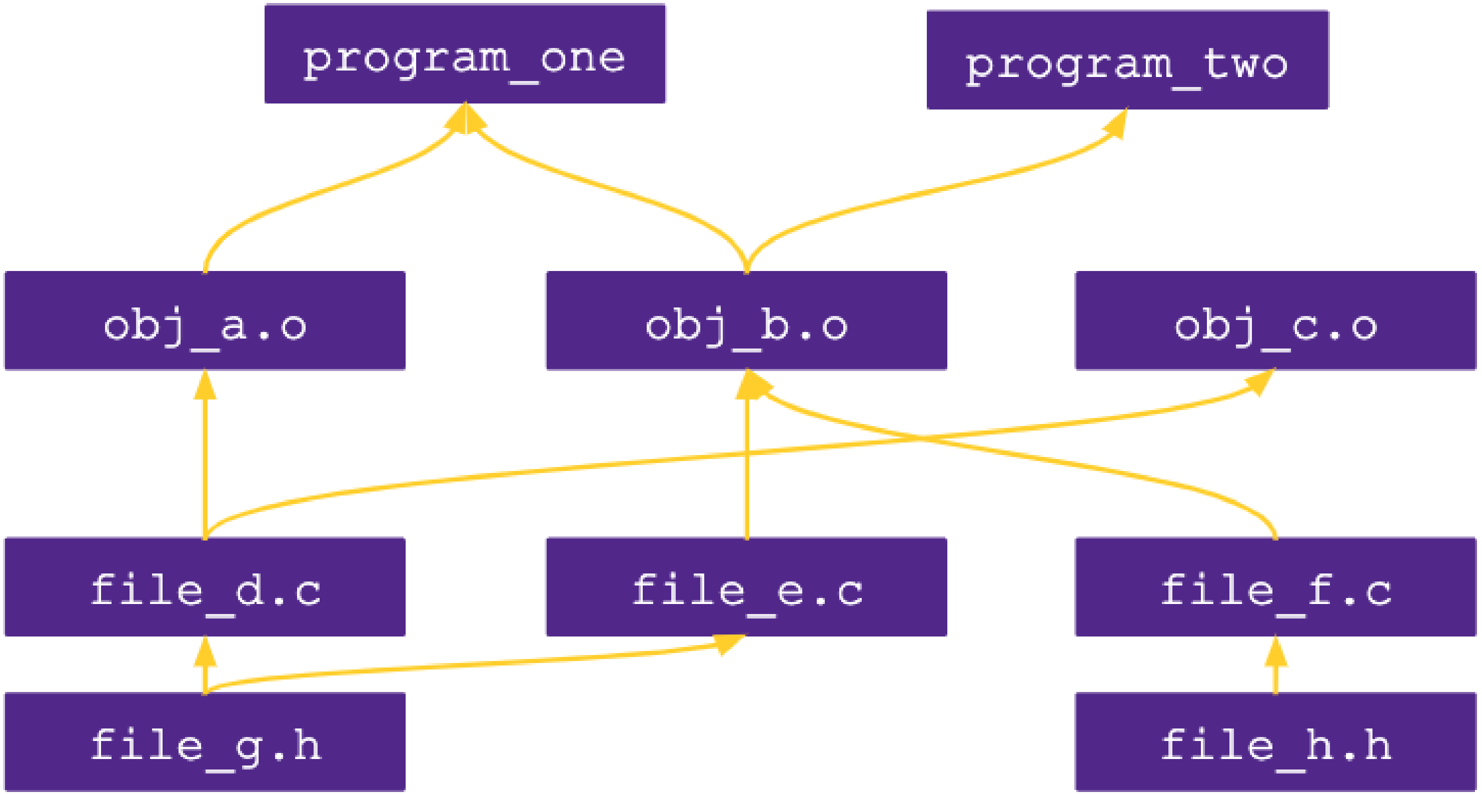
A. b, e

B. b, e, g

C. a, b ,c, e, f

D. b, e, f, g, h

E. b, d, e, f, g, h



Make Files

- Make is a program which automates building dependency trees
  - List of rules written in a Makefile declares the commands which build each intermediate part
  - Helps you avoid manually typing gcc commands, easier and less prone to typos
  - Automates build process
- Makefiles are a list of with Make rules which include:
  - Target - An output file to be generated, dependent on one or more sources
  - Source – Input source code to be built
  - Recipe - command to generate target

tab not spaces! →

```
target: source
    recipe
```

```
ll.o: ll.c ll.h
    gcc -c ll.c
```

- Makefile logic
  - Make builds based on structural organization of how code depends on other code as defined by includes
  - Recursive – if a source is also a target for other sources, must also evaluate its dependencies and remake as required
  - Make can check when you've last edited each file, and only build what is needed!
    - Files have "last modification date". make can check whether the sources are more recent than the target
  - Make isn't language specific: recipe can be any valid shell command
- run `make` command from within same folder
  - `$make [ -f makefile] [ options ] … [ targets ] ../`
  - Starts with first rule in file then follows dependency tree
  - -f specifies makefile name, if non provided will default to "Makefile"
  - if no target is specified will default to first listed in file

https://www.gnu.org/software/make/manual/make.html#Introduction

More Make Tools

- make variables help reduce repetitive typing and make alterations easier
  - can change variables from command line
  - enables us to reuse Makefiles on new projects
  - can use conditionals to choose variable settings
- ifdef checks if a given variable is defined for conditional execution
  - ifndef checks if a given variable is NOT defined
- Special characters:
  - $@ for target
  - $^ for all sources
  - $< for left-most source
  - \ enables multiline recipies
  - * functions as wildcard (use carefully)
  - % enables implicit rule definition by using % as a make specific wildcard

```
CC = gcc
CGLAGS = -Wall

foo.o: foo.c foo.h bar.h
    $(CC) $(CFLAGS) -c foo.c -o foo.o

make CFLAGS=-g

EXE=
ifdef WINDIR #defined on Windows
   EXE=.exe
endif
widget$(EXE): foo.o bar.o
    $(CC) $(CFLAGS) -o widget$(EXE)\
        foo.o bar.o

OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
    gcc -o widget $(OBJFILES)

%.o: %.c
    $(CC) -c $(CFLAGS) $< -o $@
clean:
   rm *.o widget
```

Makefile

http://cslibrary.stanford.edu/107/UnixProgrammingTools.pdf

Phony Targets

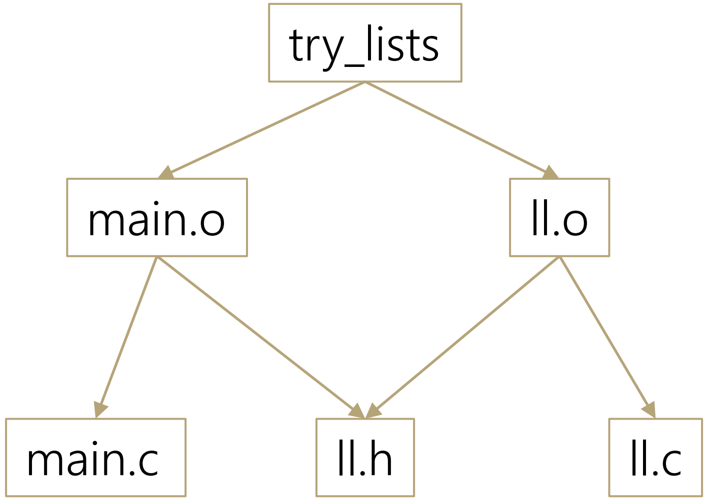
- A target that doesn't create the listed output
- A way to force run commands regardless of dependency tree
- Common uses:
  - all – used to list all top notes across multiple dependency trees
  - clean – cleans up files after usage
  - test – specifies test functionality
  - printing messages or info

```
all: try_lists test_suite
clean:
    rm objectfiles
test: test_suite
    ./test_suite
```

```
CC = gcc
CGLAGS = -Wall

all: my_program your_program

my_program: foo.o bar.o
    $(CC) $(CFLAGS) -o my_program foo.o bar.o

your_program: bar.o baz.o
    $(CC) $(CFLAGS) -o your_program foo.o baz.o

#not shown: foo.o, bar.o, baz.o targets

clean:
    rm *.o my_program your_program
```

Makefile

Example Makefile



variable definitions

must include rules for each file

rules define dependency hierarchy

```
CC = gcc
CGLAGS = -g -Wall -std=c11

try_lists: main.o ll.o
    $(CC) $(CFLAGS) -o try_lists main.o ll.o

main.o: main.c ll.h
    $(CC) $(CFLAGS) -c main.c

ll.o: ll.c ll.h
    $(CC) $(CFILES) -c ll.c
```

Makefile

Example

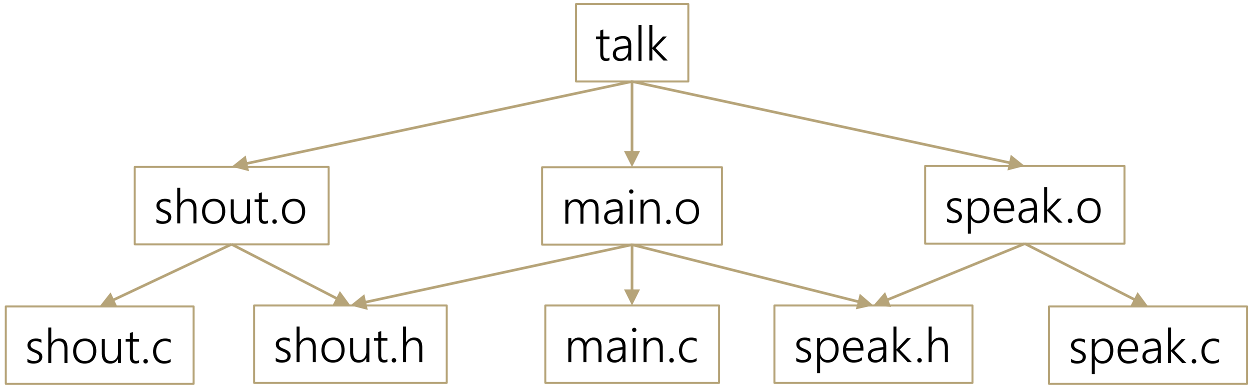
```
#ifndef SHOUT_H
#define SHOUT_H
/* Write message m in uppercase to stdout */
void shout(char m[]);
#endif /* ifndef SHOUT_H */
```
shout.h

```
#ifndef SPEAK_H
#define SPEAK_H
/* Write message m to stdout */
void speak(char m[]);
#endif /* ifndef SPEAK_H */
```
speak.h

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "speak.h"
#include "shout.h"
/* Write message m in uppercase to stdout */
void shout(char m[])
{
   int len; /* message length */
   char *mcopy; /* copy of original message */
   int i;
   len = strlen(m);
   mcopy = (char *)malloc(len*sizeof(char)+1);
   strcpy(mcopy,m);
   for (i = 0; i < len; i++)
      mcopy[i] = toupper(mcopy[i]);
   speak(mcopy); free(mcopy);
}
```
shout.c

```
#include "speak.h"
#include "shout.h"
/* Say HELLO and goodbye */
int main(int argc, char* argv[])
{
   shout("hello");
   speak("goodbye");
   return 0;
}
```
main.c

```
#include <stdio.h>
#include "speak.h"
/* Write message m to stdout */
void speak(char m[])
{
   printf("%s\n", m);
}
```
speak.c




```
all: talk
# The executable
talk: main.o speak.o shout.o
   gcc -Wall -std=c11 -g -o talk main.o speak.o shout.o

# Individual source files
speak.o: speak.c speak.h
   gcc -Wall -std=c11 -g -c speak.c
shout.o: shout.c shout.h speak.h
   gcc -Wall -std=c11 -g -c shout.c
main.o: main.c speak.h shout.h
   gcc -Wall -std=c11 -g -c main.c

# A "phony" target to remove built files and backups
clean: rm -f *.o talk *~
```
Makefile

Appendix

Extra Characters

- **In commands (short list):**
  - **$@ for target**
  - **$^ for all sources**
  - **$< for left-most source**
- **Examples:**
  - **widget$(EXE): foo.o bar.o**
    $(CC) $(CFLAGS) -o
    $@ $^
  - **foo.o: foo.c foo.h bar.h**
    $(CC) $(CFLAGS) -c $<

Also use wild cards (ex. *.o), but you need to be careful.

Use the ‘wildcard’ function for precision.

`$(wildcard *.o)`

https://www.gnu.org/software/make/manual/html_node/Wildcard-Function.html#Wildcard-Function

Fancy Stuff *(use with care!)*

Implicit rules:
Make automatically applies rules to common types of files
`n.o is made automatically from n.c with a recipe of the form '$(CC) $(CPPFLAGS) $(CFLAGS) -c'.`

Pattern rules:
Define new implicit rules by using '%' as a type of wildcard

```
%.o : %.c
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

%.class: %.java
    javac $<    # Note we need $< here
```

Commands can be any valid shell command, including shell scripts

Repeating targets can add dependencies (useful for automatic target generation)

Suffix rules:
Old form of pattern rules using only suffixes

Problem of multiple 'main' functions

```
//sample.c
#ifdef WIN32
int main() {
     //in this case only this main()
       will be compiled.
}
#endif

#ifdef LINUX
int main() {
  //another main for linux platform
}
#endif

# sample Makefile
ifdef WINDIR # defined on Windows
   CFLAGS += -D WIN32
endif
```

You would not use two 'main' functions, because main is always the single entry point.

(Note: It works in Java because we can define one 'main' for each class namespace. We don't have the same concept of namespaces in C.)

Your code could define two mains, and choose one at pre-process time.

You could also include code that was chosen with a compiler flag (such as `#ifdef DEBUG`).